CSc 656 Project 2: Cache simulation

due Thursday 5/1/2017, 3pm (no late submissions)
(8% of your grade)

This is an individual project. Work on your own.

For this project, you will write a cache simulator that uses memory address traces as input. The traces can be found on unixlab.sfsu.edu at

~whsu/csc656/Traces/MiBench

You can also access it through a web browser:

http://unixlab.sfsu.edu/~whsu/csc656/Traces

Sample executables for this project will be found on unixlab.sfsu.edu at

~whsu/csc656/Code/S17/P2

A few lines from a sample trace file:

```
0xb7fc7489: W 0xbff20468 4 0xb7fc748e
0xb7fc748e: R 0xbff20468 4 0xb7fc748e
0xb7fc7495: W 0xbff20478 4 0xbff204b0
0xb7fc749e: R 0xb7fd9ff4 4 0x15f24
```
etc etc

The first hexadecimal integer is the address of the instruction that caused the data memory access. Then follows W for write, or R for read. The next hexadecimal integer is the memory address for that data access. Then follows the number of bytes read or written. Finally, the data read/written is displayed. In the example shown, for the top line, an instruction at 0xb7fc7489 caused a data write to address 0xbff20468 of 4 bytes; 0xb7fc748e was written to memory.

Benchmark Traces

Trace excerpts (100,000 accesses each) from some benchmarks have been chosen from the Mibench embedded applications suite. Each student will be assigned two traces. These are:

from consumer suite: lame, mad, jpeg
from telecom suite: gsm, adpcm, FFT
from security suite: sha, blowfish
from network suite: dijstra, patricia

You should make all measurements for each cache configuration, using the trace for the

benchmark.

Cache simulator implementation

You will write a cache simulator to collect statistics on events in the memory system. Your code will allocate some arrays that represent the state of the cache, read each line of a trace file in the specified format, and update the state of the proper cache array elements accordingly.

While you are only required to show measurements for a small number of cache configurations, your code *must* work for any cache size that is a power of 2 bytes.

Your simulator tracks the behavior of a writeback data cache, as defined below.

Cache System Characteristics

For both benchmarks, study cache configurations with 16-byte blocks. Make measurements for these systems:

System 1: Direct-mapped data cache, 1KB and 2KB
System 2: k-way set associative data cache, 1KB and 2KB

Make measurements for each benchmark, for the following parameters:

System	Cache configuration
System 1a	1KB direct-mapped
System 1b	2KB direct-mapped
System 2a	1KB 2-way set associative
System 2b	1KB 4-way set associative
System 2c	2KB 2-way set associative
System 2d	2KB 4-way set associative

Assume that all read hits take 1 cycle.
Assume that the miss penalty is equal to 80 cycles.
On a write miss, assume that the requested block is always brought into the cache, and the CPU writes into the proper word in the cache.

You should collect these statistics, and display them in order (see sample executables):

- number of data reads (i.e., number of loads),
- number of data writes (i.e., number of stores),
- number of data accesses (i.e., number of loads + number of stores),
- number of total data read misses,
- number of total data write misses,
- number of data misses (i.e., number of accesses that miss the cache),
- number of dirty data read misses,
- number of dirty write misses,

- number of bytes read from memory,
- number of bytes written to memory,
- the total access time (in cycles) for reads,
- the total access time (in cycles) for writes,
- the overall data cache miss rate ((read misses + write misses) / total accesses)

A data read miss is any read access (i.e., load) that results in an access to memory.
A data write miss is any write access (i.e., store) that results in an access to memory.
A clean miss is a miss that does not replace a dirty block in the cache (i.e., the dirty bit for the block is off, or the block is not valid).
A dirty miss is a miss that replaces a dirty block in the cache.
(Note that some of these measurements give overlapping information, and it is easy to check whether your results are consistent by comparing some of the measurements.)

Follow the definitions given below for detailed operations, and how hits and misses are counted.

Detailed direct-mapped data cache operation (System 1)

Suppose an access has address A. A maps to index I in the direct-mapped data cache. Each access falls under one of these cases:

Case 1: the block containing A is found in data cache (***cache hit***)
Read: no state changes, *1 cycle*
Write: set dirty bit = 1, *1 cycle*

[Case 2a/2b cover misses. For a cache miss, index I in the data cache may contain block X (different from the block that contains address A), or may be empty.]

Case 2a: the block containing A is not found in data cache, index I in data cache either contains block X (which is clean), or is empty (***clean cache miss***)
Read: move block containing A from memory into index I in data cache,
dirty bit = 0, *(1 + miss penalty) cycles*
Write: move block containing A from memory into index I data cache, dirty bit = 1
(1 + miss penalty) cycles

Case 2b: the block containing A is not found in data cache, index I in data cache contains block X in data cache, which is dirty (***dirty cache miss***)
Read: write block X to memory
move block containing A from memory into data cache, dirty bit = 0
*(1 + 2 * miss penalty) cycles*
Write: write block X to memory
move block containing A from memory into data cache, dirty bit = 1
*(1 + 2 * miss penalty) cycles*

Detailed k-way set-associative data cache operation (System 2)

All cases are basically the same as for System 1, except the cache is k-way set-associative.

Suppose an access has address A. A maps to index I in the direct-mapped data cache. Each index I in the data cache accesses one of k blocks. Assign ids 0 to k-1 to the k blocks within the same index.

For each block in the k-way set-associative cache, in addition to the bit-fields in the direct-mapped cache, there is also a *last-used* field. This contains the current instruction count of the access that last touched that block, i.e., the order of the access in the trace file, starting from 0.

Index I in the data cache contains block X_0 to X_{k-1} , corresponding to ids 0 to k-1; each of these k blocks is either different from the block that contains address A, or is empty.

Each access falls under one of these cases:

Case 1: the block containing A is found in index I id D in the data cache (***cache hit***)
Read: last-used for index I id D = current IC, *1 cycle*
Write: last-used for index I id D = current IC, set dirty bit = 1, *1 cycle*

[Case 2a/2b cover misses. For a cache miss, the block containing A maps to index I. First choose one of k blocks. If there is an empty block, this is the empty block with the smallest id. If there is no empty block, this is the least recently used block, according to the *last-used* field for that block. Let the id of the chosen block be D.]

Case 2a: the block containing A is not found in data cache, the chosen block with id D is either clean, or is empty (***clean cache miss***)
Read: move block containing A from memory into block D in data cache,
last-used for index I id D = current IC, dirty bit = 0
(1 + miss penalty) cycles
Write: move block containing A from memory into index I data cache
last-used for index I id D = current IC, dirty bit = 1
(1 + miss penalty) cycles

Case 2b: the block containing A is not found in data cache, the chosen block with id D is dirty (***dirty cache miss***)
Read: write block X to memory
move block containing A from memory into block D in data cache,
last-used for index I id D = current IC, dirty bit = 0
*(1 + 2 * miss penalty) cycles*
Write: write block X to memory
move block containing A from memory into block D in data cache,
last-used for index I id D = current IC, dirty bit = 1

*(1 + 2 * miss penalty) cycles*

Command line format

Your simulators will follow this command line format. For sys1:

`./sys1 tracefile cachesize [-v ic1 ic2]`

`tracefile` is the name of the trace file, *`cachesize`* is the size of the cache in KB, and –v is the optional verbose flag. Verbose output format will be defined in the next section.

Sys2 has similar command line format as sys1, but with an extra argument for set associativity k:

`./sys2 tracefile cachesize set-associativity [-v ic1 ic2]`

Verbose format

The –v flag is always followed by two integers ic1 and ic2. The data memory accesses in the trace file are numbered in sequential order, starting from zero. Verbose output is displayed for access ic1 to ic2, inclusive.

For System 1 (direct-mapped cache), the verbose format output comprises:

order of the load/store in address stream (starting from 0, in decimal)
cache index calculated from address
cache tag calculated from address
valid bit of cache block accessed
tag stored in cache block accessed (0 if invalid)
dirty bit of cache block accessed
hit or miss (0 miss, 1 hit)
Case number according to Detailed Operation (1, 2a, or 2b)

For example, reference 0 from adpcm.xex for a 1KB direct-mapped cache looks like this:

0 36 2feee4 0 0 0 0 2a

The original reference address is 0xbfbb9368, which breaks up into tag = 2feee4 and index = 36. The cache was empty, so the valid bit is 0, the old tag and dirty bits are also 0. It's a miss (0), and this is Case 2a (clean miss with empty cache block).

For System 2 (k-way set-associative cache), the verbose format comprises:

order of the load/store in address stream (starting from 0, in decimal)
cache index calculated from address

cache tag calculated from address
valid bit of cache block accessed
id of the block chosen (D, from the specification)
last-used field of the block chosen (0 if invalid)
tag stored in cache block accessed (0 if invalid)
dirty bit of cache block accessed
hit or miss (0 miss, 1 hit)
Case number according to Detailed Operation (1, 2a, or 2b)

For example, consider three references from adpcm.xex for a 1KB 2-way set-associative cache:

343 4 40592 1 0 308 40592 1 1 1 (original address 0x80b244f)
881 4 40570 1 1 846 40570 0 1 1 (original address 0x80ae04c)
911 4 40590 1 0 343 40592 1 0 2b (original address 0x80b2040)

Reference 343 has index 4, tag 0x40592. It is found in a valid block (1), with id 0. That block was last touched by reference 308, the stored tag is 0x40592 (since it's a hit), it was dirty (1), we have a hit (1), and Case 1.

Reference 881 has index 4, tag 0x40570. It is found in a valid block (1), with id 1. That block was last touched by reference 846, the stored tag is 0x40570 (since it's a hit), it was clean (0), we have a hit (1), and Case 1.

Reference 911 has index 4, tag 0x40590. It is not found in either of the two blocks with index 4. The LRU block is the one with id 0, which was last touched by reference 343. Hence we choose for it a valid block (1), with id 0. That block was last touched by reference 343, the stored tag is 0x40592, it was dirty (1), we have a hit (1), and Case 2b (dirty miss).

Submission (2 parts):

Source code submission: submit a single tar, zip or other archive file using iLearn. Your archive file should expand into a single directory tagged with your login name. The directory should contain your source files and a makefile.

I should be able to cd into the directory you submitted, type

```
make sys1
make sys2
```

to generate two executables, sys1 (for System 1) and sys2 (for System 2). I will then run each one with this command line:

```
sys1 tracefile 1
sys1 tracefile 2
sys2 tracefile 1 2 etc etc
```

If you don't provide makefiles, you *must* include a README file that includes step-by-step instructions for compiling your source code.

You may write your cache simulator in Java, with or without an IDE. However, you *must* include a README file that includes clear step-by-step instructions for compiling and running your code.

If you would like to deviate from the specified command line interface, you must get my permission before Monday 4/17 11am.

Report submission: Submit a table (in pdf format) showing *miss rate* and *total access time* for each system, for each benchmark you are responsible for. Your table should look something like

	Benchmark 1 miss rate	Benchmark 1 tot. access time	Benchmark 2 miss rate	Benchmark 2 tot. access time
System 1a				
System 1b				
System 2a				
System 2b				
System 2c				
System 2d				

Note: The address traces used in this project are generated using the Pin program instrumentation toolkit, using the *atrace* or address trace tool. While you don't need to know anything about Pin, if you're curious, check out

http://www.pintool.org/

The Pin toolkit and atrace tool can also be found on thecity.sfsu.edu in

~hsu/ pin-2.9-39599-gcc.3.4.6-ia32_intel64-linux/SimpleExamples/pinatrace